CS143 Compilers - Written Assignment 3

Due Tuesday, May 23, 2017 at 11:59 PM

This assignment covers semantic analysis, including scoping and type systems. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work, and you should indicate in your submission who you worked with, if applicable. Assignments can be submitted electronically through Gradescope as a PDF by Tuesday, May 23, 2017 11:59 PM PDT. A LaTeX template for writing your solutions is available on the course website.

1. **(9 pts)** The following is the implementation of the Main class of a cool program:

```
class Main {
  b : B;
  main() : Int {{
    b <- new B;
    b.h();
    2;
  }};
};
```

Now consider the following implementations of the classes A and B. Analyze each version of the classes to determine if the resulting program will pass type checking and, if it does, whether it will execute without runtime errors. Please include a brief (1 - 2 sentences) explanation along with your answer.

(a) **(3 pts)**

```
class A {
  i : Int;
  a : SELF_TYPE;
};

class B inherits A {
  h() : A {
    if (i <- 1) < 2 then a <- new A else a <- new B fi
  };
};
```

(b) **(3 pts)**

```
class A {
  i : Int;
  a : A;
};

class B inherits A {
  h() : A {
    case a of
      k: A => a;
      b: B => b;
      o: Object => a;
    esac
  };
};
```

(c) **(3 pts)**

```
class A {
  i : Int;
  a : A;
  d : SELF_TYPE;
};

class B inherits A {
  h() : A {{
    d <- self;
    case d of
      k: A => d;
      b: B => d;
      o: Object => d;
    esac;
  }};
};
```

2. **(8 pts)**

(a) **(4 pts)** Consider the following program in Cool (using standard Cool type rules, scoping rules and general semantics). Provide the output of the the labelled statement in Main.main() and explain why it prints that value.

```
class A {
  f1(): Int {
    let a: Int in {
        a <- x + 1;
        a;
    }
  };
  f2(): Int {
    let a: Int in {
      { a <- 2; };
      x <- a;
      f1();
      a + x;
    }
  };
  x: Int <- 0;
  a: Int <- 1;
};
class Main {
  main(): Object {
    let o: A, io: IO <- New IO in {
      o <- new A;
      io.out_int(o.f2()); -- Statement
    }
  };
};
```

(b) **(4 pts)** In the following program, suppose [Placeholder B] will be filled by an integer literal that is unknown to you. Can you replace [Placeholder A] with a Cool expression that will allow you to predict the output of the labelled statement? If you are able to do so, provide your replacement for [Placeholder A] and the output of the statement. If you cannot, explain why.

```
class Main {
  main(): Object {
    let io: IO <- New IO, z: Int (* <- [Placeholder B] *) in {
      let w: Int <- 2 in {
        let z: Int <- w in {
          (* [Placeholder A] *)
          io.out_string("The secret will be: ");
          io.out_int(z);
        };
      };
      io.out_string("The secret is: ");
      io.out_int(z); -- Statement
    }
  };
};
```

3. **(12 pts)** Type derivations are expressed as inductive proofs in the form of trees of logical expressions. For example, the following is the type derivation for $O[Int/y] \vdash y + y : Int$:

$$\frac{\dfrac{O[Int/y](y) = Int}{O[Int/y], M, C \vdash y : Int} \qquad \dfrac{O[Int/y](y) = Int}{O[Int/y], M, C \vdash y : Int}}{O[Int/y], M, C \vdash y + y : Int}$$

Consider the following Cool program fragment:

```
class A {
  a: Int;
  b: Int;
  c: Int;
  yes: Bool;
  foo(): SELF_TYPE { self };
  bar(k : A) : Bool { if a < 0 then yes else false fi };
};
class B inherits A {
  me: SELF_TYPE;
  test(): Object { (* [Placeholder] *) };
};
```

Note that the environments O and M at the start of the method test(...) are as follows:

$$O = \emptyset[\text{Int}/a][\text{Int}/b][\text{Int}/c][\text{Bool}/yes][\text{SELF\_TYPE}_B/me]$$

M(A, foo) = (SELF_TYPE)
M(A, bar) = (A, Bool)
M(B, foo) = (SELF_TYPE)
M(B, bar) = (A, Bool)
M(B, test) = (Object)

For each of the following expressions replacing [Placeholder], provide the type derivation and final type of the expression, if it is well typed, otherwise explain why it isn't. Assume Cool type rules (you may omit subtyping relationships from the rules when the type is the same, e.g. Bool $\leq$ Bool).

(a)

```
a <- c + { if a < b then a else b fi; }
```

(b)

```
me.bar(me.foo())
```

(c)

```
me.bar(me <- new B)
```

4. **(16 pts)** Consider the following extension to the Cool syntax as given on page 16 of the Cool Manual, which adds arrays to the language:

$$\begin{aligned} \text{expr} ::= &\ \text{new TYPE[expr]} \\ &\mid \text{expr[expr]} \\ &\mid \text{expr[expr]}\ <-\ \text{expr} \end{aligned} \tag{1}$$

This adds a new type T[] for every type T in Cool, including the basic classes. Note that the entire hierarchy of array types still has Object as its topmost supertype. An array object can be initialized with an expression similar to "my_array:T[] ← new T[n]", where n is an Int indicating the size of the array. In the general case, any expression that evaluates to an Int can be used in place of n. Thereafter, elements in the array can be accessed as "my_array[i]" and modified using a expression like "my_array[i] ← value".

(a) **(4 pts)** Provide new typing rules for Cool which handle the typing judgments for: $O, M, C \vdash \text{new T[} e_1 \text{]}$, $O, M, C \vdash e_1[\ e_2\ ]$ and $O, M, C \vdash e_1[\ e_2\ ]\ <-\ e_3$. The type of the expression $e1[e2]\ <-\ e3$, if well-typed, should be the type of e3. Make sure your rules work with subtyping.

(b) **(4 pts)** Consider the following subtyping rule for arrays:

$$\frac{T_1 \leq T_2}{T_1[\,] \leq T_2[\,]}$$

This rule means that $T_1[\,] \leq T_2[\,]$ whenever it is the case that $T_1 \leq T_2$, for any pair of types T_1 and T_2.

While plausible on first sight, the rule above is incorrect, in the sense that it doesn't preserve Cool's type safety guarantees. Provide an example of a Cool program (with arrays added) which would type check when adding the above rule to Cool's existing type rules, yet lead to a type error at runtime. The error should be related to the rules for arrays and not be among the kinds of runtime errors defined on page 29 of the Cool manual.

(c) **(4 pts)** Suppose you wish to extend the subtyping rule as far as possible while preserving the soundness of the typesystem. What are all types X such that $T[] \leq X$? What are all types Y such that $Y \leq T[]$?

(d) **(4 pts)** Add another extension to the language for immutable arrays (denoted by the type T()). Immutable arrays are initialized by providing a list of expressions for the array to hold. After initialization, the contents of an immutable array may not be modified. Analogous to questions 4a and 4c, for this extension, provide: the additional syntax constructs to be added to the listing of page 16 of the Cool manual, the typing rules for these constructs and the least restrictive subtyping relationship involving these tuple types. It is not necessary that this extension interact correctly with mutable arrays as defined above, but feel free to consider that situation.

5. **(8 pts)** Consider the following assembly language used to program a stack machine (r, r1, r2 denote arbitrary registers):

```
1   push r: pushes the value in r onto the stack
2   top r: copies the value at the top of the stack into r. This
       command does not modify the stack.
3   pop: discards the value at the top of the stack
4   r1 += r2: adds r1 and r2 and saves the result in r1. r1 may be the
        same as r2.
5   r1 -= r2: subtracts r2 from r1 and saves the result in r. r1 may
       be the same as r2.
6   jump r: jumps to the line number in r and resumes execution
7   print r: prints the value in r to the console
```

The machine has three registers available to the program: reg1, reg2, and reg3. The stack is permitted to grow to a finite, but very large, size. If an invalid line number is invoked, pop is executed on an empty stack, or the maximum stack size is exceeded, the machine crashes.

(a) **(4 pts)** Write code to enumerate the Fibonnacci series (0, 1, 1, 2, ...) without termination. Assume that the code will be placed at line 100, and will be invoked by setting reg1, reg2, and reg3 to 100, 0, 1 respectively and running 'jump reg1'. Your code should use the 'print' opcode to display numbers in the series.

(b) **(4 pts)** This 'helper' function is placed at line 1000:

```
1000    push reg1
1001    reg1 -= reg2
1002    reg2 -= reg1
1003    reg3 += reg2
1004    top reg1
1005    pop
1006    jump reg1
```

This 'main' procedure is placed at line 2000:

```
2000    push reg1
2001    push reg3
2002    top reg1
2003    top reg3
2004    pop
2005    pop
2006    jump reg2
2007    print reg3
2008    jump reg2
```

reg1, reg2, reg3 are set to 0, 1000 and 2000 respectively, and 'jump reg3' is executed. What output does the program generate? Does it crash? If it does, suggest a one-line change to the helper function that results in a program that does not crash.